Ambiente para la simulación de distintos ACP y recuperación de errores en Bases de Datos Distribuidas

A.C. Sebastián Ruscuni[1]
Lic. Rodolfo Bertone [2]

Laboratorio de Investigación y Desarrollo en Informática[3]
Facultad de Informática
UNLP

Palabras Clave: sistemas distribuidos, bases de datos distribuidas, simulación, integridad de datos, protocolos

Resumen

Este artículo presenta una evolución de un ambiente de simulación donde se modelizan e implementan situaciones en la que una BDD debe mantener la integridad de la información ante fallos producidos durante la ejecución de transacciones, enfocándose principalmente en la utilización y posterior comparación de diferentes protocolos de cometido atómicos (ACP). La implementación se realiazó en Java debido a diferentes aspectos como facilidad de trabajo, portabilidad, etc..

El ambiente se basa principalmente en la recuperación de fallas de transacciones en un entorno de datos distribuidos, permitiendo seleccionar, para el desarrollo de la simulación, entre los protocolos dos fases, de tres fases, el protocolo optimista y pesimista. Se especifica, además, el schedule de tareas definido e implementado para la realización de cada simulación a partir de una traza de ejecución que establece cada problema puntual a resolver.

[1] Analista en Computación. Alumno avanzado de la Licenciatura en Informática. Facultad de Informática, UNLP.
Ayudante Diplomado con Dedicación Semi-Exclusiva.
E-mail sruscuni@lidi.info.unlp.edu.ar
[2] Prof. Adjunto con Dedicación Exclusiva, LIDI. Facultad de Informática, UNLP.
E-mail pbertone@lidi.info.unlp.edu.ar
[3] Calle 50 y 115 Primer Piso, (1900) La Plata, Argentina, Teléfono +54 221 422 7707
WEB: lidi.info.unlp.edu.ar

Introducción

Se define un sistema distribuido como una colección de computadoras autónomas interconectadas mediante una red, con un software diseñado para brindar facilidades integradas de cómputo. Los sistemas distribuidos están implementados sobre plataformas de hardware que varían en su tamaño y conexión.

Las características claves de un sistema distribuido son: soportes para compartir recursos, concurrencia, escalabilidad, tolerancia a fallos y transparencia. [COUL95]

Hay varias razones para desarrollar y usar un sistema de bases de datos distribuidas: [HANS97]

- Las organizaciones tienen divisiones en diferentes localidades. Para cada localidad puede haber un conjunto de datos que usan frecuente y exclusivamente.
- Permitir que cada sitio almacene y mantenga su propia base de datos local facilita el acceso inmediato y eficaz a los datos que se usan con más frecuencia.
- Las bases de datos distribuidas logran mejoras en la fiabilidad del sistema, aunque agregan otros elementos que hacen al mantenimiento de seguridad e integridad de información.

Una Base de Datos Distribuida puede ser definida como una colección integrada de datos compartidos que están físicamente repartidos a lo largo de los nodos de una red de computadoras. Un DDBMS es el software necesario para manejar una BDD de manera que sea transparente para el usuario. [BURL 94]

En un sistema de base de datos distribuida, los datos se almacenan en varios computadores. Las estaciones de trabajo de un sistema distribuido se comunican entre sí a través de diversos medios, tales como cables de alta velocidad o líneas telefónicas. No comparten la memoria principal ni el reloj.

La tecnología general de BDD involucra dos conceptos diferentes, llamados **integración** a través de los elementos que componen una base de datos y **distribución** a través de los elementos de una red. La **distribución** es provista al repartir los datos a través de los diferentes sitios de la red, mientras que la **integración** está dada a partir del agrupamiento lógico de los datos distribuidos haciéndolos aparecer como una simple unidad ante la vista del usuario final de la BDD.

Un DBMS centralizado es un sistema que maneja una BD simple, mientras que un DDBMS es un DBMS simple que maneja múltiples BD. El término global y local se utiliza, cuando se discute sobre DDBMS, para distinguir entre aspectos que se refieren al sitio simple (local) y aquello que se refiere al sistema como un todo (global). La BD local se refiere a la BD almacenada en un sitio de la red, mientras que la DB global se refiere a la integración lógica de todas las BD locales. [BELL 92]

La utilización de BDD tiene asociado una serie de beneficios y costos que deben evaluarse en el momento de tomar decisiones respecto de su utilización.[ÖZSU 91]. Algunas de las ventajas mencionables son:

- Autonomía local: cada sitio de la red en un entorno distribuido debe ser independiente del resto. Cada BD local es procesada por su propio DBMS.
- Mejoras en la performance.
- Mejoras en la disponibilidad de la información.
- Economía: el equipamiento necesario para montar una red para distribuir la información, es comparativamente mucho menor que tener un sitio central

con gran capacidad de procesamiento (asociado con el concepto de downsizing) [UMAR 93]
- Sistemas fácilmente expandibles.
- Compartir información.

Entre los inconvenientes y desventajas que surgen están:

- Menor experiencia en el desarrollo de sistemas y BD distribuidas
- Complejidad
- Costo: se requiere hardware adicional, software más complejo, y el costo del canal de transmisión, para conectar los nodos distribuidos de la red.
- Distribución de control: se contradice con una de las ventajas planteadas. Se hace hincapié, en este punto, en los problemas creados por la sincronización y coordinación de tareas.
- Seguridad: aparecen nuevos problemas, y más complejos relacionados con la seguridad de la información manipulada.

Transacciones, integridad y seguridad

Una transacción es una colección de operaciones que realiza una única función lógica. Cada transacción es una unidad de atomicidad (debe ocurrir completa o no ocurrir). Los algoritmos para asegurar la consistencia de la BD, utilizando las transacciones, incluyen: [SILB 98]

- Acciones tomadas durante el procesamiento normal de las transacciones que aseguran la existencia de información suficiente para asegurar la recuperación de fallos.
- Acciones tomadas a continuación de un fallo, para asegurar la consistencia de la BD.

Las transacciones transforman la base de datos de un estado consistente hacia otro también consistente. Pero se debe tener en cuenta que la consistencia de la base de datos puede ser violada durante la ejecución de la transacción. Las cuatro propiedades básicas de una transacción, denominadas **A.C.I.D.**, son:

1. Atomicidad: la propiedad del “todo o nada”; una transacción es una unidad indivisible
2. Consistencia: las transacciones transforman la base de datos de un estado consistente a otro también consistente
3. Independencia: las transacciones se ejecutan independientemente una de las otras (los efectos parciales de una transacción incompleta no son visibles para el resto de las transacciones)
4. Durabilidad (también llamada persistencia): los efectos de transacciones que ya fueron completadas (cometidas) son almacenados permanentemente en la base de datos y no pueden ser desechos.

En un ambiente de base de datos distribuida, una transacción puede acceder a datos que están almacenados en más de un sitio de la red. Cada transacción es dividida en una serie de sub-transacciones, una por cada sitio en donde existan datos que la transacción original necesita procesar.

Existen dos mecanismos utilizados para asegurar la atomicidad de las transacciones:

- Basado en bitácora
- Doble paginación

Protocolos de Commit Atómicos (ACP)

Un modelo común para una transacción distribuida se centra en un proceso, llamado *coordinador*, que se ejecuta en el sitio en donde la transacción se crea, y un conjunto de procesos, llamados *localidades*, que se ejecutan en distintos sitios que deben ser accedidos por la transacción.

Para garantizar la atomicidad, es preciso que en todas las localidades en las que se haya ejecutado la transacción distribuida *T* coincidan en el resultado final de la ejecución. *T* debe quedar ejecutada o abortada en todas las *localidades*. Para garantizar esta propiedad, el *coordinador* de transacciones encargado de *T* debe ejecutar un protocolo de commit.

En las últimas dos décadas, investigadores de bases de datos han propuesto una gran variedad de protocolos. Entre los más importantes incluimos al *protocolo de dos fases (2PC)*, dos variaciones de éste llamadas *protocolo pesimista (PrA) y optimista (PrC)*, y por último al *protocolo de tres fases (3PC)*. Según la funcionalidad de cada uno, los protocolos de commit requieren intercambiar mensajes, a través de distintas fases, entre sitios participantes donde la transacción distribuida es ejecutada. Por lo tanto, son generados varios registros en memoria estable, alguno de los cuales son grabados a disco de manera sincrónica. A causa de estos costos adicionales, el procesamiento de commit puede resultar en un incremento del tiempo de ejecución de la transacción, haciendo que la elección del protocolo sea una importantísima decisión de diseño para sistemas de bases de datos distribuidas.

Protocolo de dos fases

Cada transacción se genera en una localidad, el coordinador local de transacciones es el encargado de controlar la ejecución de la misma. En caso de ser una transacción local, se procede igual a los sistemas centralizados. Ahora, si la transacción accede a datos globales, el coordinador debe encargarse de subdividir la transacción en subtransacciones que se ejecutarán en diferentes localidades de la red. Cada localidad participante recibe la subtransacción que puede tratar y la procesa como si ésta fuera una local. [WEB1] [WEB2].

A continuación se describen brevemente las dos fases del protocolo:

- **Fase 1**: el coordinador envía un mensaje a cada localidad involucrada preguntando si se pudo finalizar la transacción. Ante una respuesta negativa, o eventualmente, falta de respuesta, la transacción debe ser abortada. Si todos contestan afirmativamente la transacción está en condiciones de ser cometida.
- **Fase 2:** se envía un mensaje de abortar (si hay al menos una respuesta negativa o se cumple un tiempo de espera predeterminado), o un mensaje de finalizar (si todas incluyendo al coordinador pudieron finalizar la transacción. Cada localidad comete la transacción y envía un mensaje de reconocimiento al coordinador.

El protocolo que se utiliza incorpora un *timeout,* es decir, se tiene en cuenta el tiempo de espera máximo entre nodos. Si una subtransacción no recibe un mensaje del coordinador le pide a todas las participantes un mensaje de ayuda. Cuando alguna recibe este mensaje puede ayudar a su compañera, de acuerdo a su propio estado; es decir, si ha recibido del coordinador un mensaje commit o abort, envía el mismo. La transacción en espera puede ahora realizar el commit o abortar sin ningún lugar a dudas, pues su compañera ha replicado el mensaje seguro del coordinador. Por el contrario, si la subtransacción no recibe ningún mensaje, continuará en espera. [ZANC96] [BERN84]

Protocolo Pesimista

Una variante del protocolo de dos fases (2PC), llamado **protocolo pesimista**, trata de reducir el número de mensajes que se intercambian entre los participantes de la ejecución de una transacción distribuida, utilizando la regla: "en caso de duda, se aborta la transacción". Esto es, si luego de un fallo, un sitio consulta al *coordinador* por el estado de la transacción y no recibe información del mismo, se asume que la transacción abortó. Con esta suposición, no será necesario que las localidades:

a) Envíen los mensajes de reconocimiento (ACK) al coordinador por los mensajes de ABORT del mismo
b) Escriban el registro de ABORT en la bitácora. Y también no será necesario que el *coordinador* escriba el registro de ABORT ni el de END en la bitácora para una transacción que abortó.

En resumen, el protocolo pesimista es idéntico al protocolo de dos fases para transacciones que terminan cometiendo, pero reduce el número de mensajes y el overhead de almacenamiento en bitácora, para transacciones que terminan abortando.

Protocolo Optimista

Una variación del protocolo pesimista se basa en la observación que, en general, el número de transacciones que cometen es mayor al número que abortan. En este protocolo, llamado protocolo optimista, el overhead se reduce para las transacciones que cometen, en vez de para las que abortan, requiriendo a todos los participantes de la transacción distribuida que cumplan la regla que indica "en caso de duda, cometan". En este esquema, las *localidades* no envían el reconocimiento (ACK) en la decisión global de commit, y no escriben el registro de COMMIT en la bitácora. Por lo tanto, el *coordinador* tampoco escribe el registro END en memoria estable.

Protocolo de tres fases

El problema fundamental con los protocolos descriptos anteriormente es que las *localidades* podrían *bloquearse* en el caso de un fallo hasta que el sitio que falló se recupere. Por ejemplo, si el *coordinador* falla luego de iniciar el protocolo pero antes de comunicar su decisión a cada *localidad,* éstas se bloquearán hasta que el *coordinador* se recupere y les informe su decisión. Durante el tiempo en que las *localidades* están bloqueadas, continúan manteniendo recursos del sistema, como por ejemplo *locks* sobre algunos items de datos, haciendo que no estén disponibles para otras transacciones.

Para atacar el problema de bloqueo, fue propuesto el **protocolo de tres fases (3PC)**. Activa la capacidad de no bloquearse agregando una nueva fase: "precommit"

entre las dos fases definidas en el protocolo de dos fases. Aquí se obtiene una decisión preliminar, que será comunicada a todos las *localidades* participantes de la transacción, permitiendo que la decisión global del destino de la transacción se genere independientemente de un posible fallo del *coordinador*. Cabe destacar, que el precio de obtener la funcionalidad de no bloquearse, es que existirá un número mayor de mensajes que se intercambiarán entre el *coordinador* y las *localidades*, ya que existe una fase más. Por lo tanto, también necesitarán escribir registros adicionales en memoria estable, durante la fase de "precommit".

El protocolo de tres fases requiere que:

- No pueda ocurrir una fragmentación de la red.
- Debe haber al menos una localidad funcionando en cualquier punto.
- En cualquier punto, como máximo un numero *K* de participantes pueden caer simultáneamente (siendo *K* un parámetro que indica la resistencia del protocolo a fallos en localidades).

El protocolo alcanza esta propiedad de no-bloqueo añadiendo una fase extra, en la cual se toma una decisión preliminar sobre el destino de *T*. Como resultado de esta decisión, se pone en conocimiento de las localidades participantes cierta información, que permite tomar una decisión a pesar del fallo del coordinador.

- **Fase 1:** Esta fase es idéntica a la fase 1 del protocolo de commit de dos fases.
- **Fase 2:** Si *el coordinador* recibe el mensaje **abortar** *T* (siendo *T* una transacción) de una localidad participante, o si no recibe respuesta dentro de un intervalo previamente especificado de una localidad participante, entonces decide abortar *T*. La decisión de abortar está implementada de la misma forma que en protocolo de commit de dos fases. Si recibe un mensaje *T* **lista** de cada localidad participante, tomará la decisión de <<preejecutar>> *T*. La diferencia entre preejecutar y ejecutar radica en que *T* puede ser todavía abortado eventualmente. La decisión de preejecutar permite al coordinador informar a cada localidad participante que todas las localidades participantes están <<listas>>. De acuerdo a ésto, envía un mensaje **preejecutar** *T* a todas las localidades participantes. Cuando una localidad recibe un mensaje del coordinador (ya sea **abortar** o **preejecutar** *T*) envía un mensaje de **reconocimiento a** *T* al coordinador.
- **Fase 3:** Esta fase se ejecuta sólo si la decisión en la Fase 2 fue de preejecutar. Después de que los mensajes **preejecutar** *T* se han enviado a todas las localidades participantes, el coordinador debe esperar hasta que reciba al menos un número *K* de mensajes de **reconocimiento a** *T*. Siguiendo este proceso, el coordinador toma una decisión de commit. De acuerdo a ésto, envía un mensaje **ejecutar** *T* a todas las localidades participantes.

Tal y como en el protocolo de commit de dos fases, una localidad en la que se halla ejecutado *T* puede abortar *T* incondicionalmente en cualquier momento, antes de enviar el mensaje *T* lista al coordinador. El mensaje *T* **lista** es, de hecho, una promesa hecha por la localidad para seguir la orden del coordinador de ejecutar *T* o abortar *T*. En contraste con el protocolo de commit de dos fases, en el que el coordinador puede incondicionalmente abortar *T* en cualquier momento antes de enviar el mensaje ejecutar

T, el mensaje **preejecutar** *T* en el protocolo de commit de tres fases, es una promesa hecha por el coordinador para seguir la orden del participante de ejecutar *T*.

Arquitectura utilizada

Las computadoras sobre Internet están conectadas a través del protocolo TCP/IP. En la década del 80, ARPA (Advanced Research Projects Agency) del gobierno de EEUU, desarrolló una implementación bajo UNIX del protocolo TCP/IP. Allí fue creada una interface socket. En la actualidad, la interface socket constituye el método más usado para el acceso a una red TCP/IP. [WEB4]

Un socket es una abstracción que representa un enlace punto a punto entre dos programas ejecutándose sobre una red TCP/IP. Utiliza el modelo Cliente/Servidor. Cuando dos computadoras desean conversar, cada una usa un socket. Una de ellas es el "Server" que abre el socket y espera por alguna conexión. La otra es el "Cliente" que llamará al socket server para iniciarla. Además, para establecer la conexión, solo será necesaria la dirección del Server y el número de puerto que se utilizará para la comunicación. Aquí se utiliza principalmente el package java.net. Cuando los dos sockets están comunicados, el intercambio de datos se realiza mediante InputStreams y OutputStreams. [WEB5]

La ventaja de este modelo sobre otros tipos de comunicación se ve reflejada en que el Server no necesita tener ningún conocimiento sobre el sitio en donde reside el cliente. Por otra parte, plataformas como UNIX, DOS, Macintosh o Windows no ofrecen ninguna restricción para la realización de la comunicación. Cualquier tipo de computadora que soporte el protocolo TCP/IP podrá comunicarse con otra que también lo soporte a través del modelo de sockets. [WEB6]

Presentación del trabajo

Se realizaron simulaciones sobre la ejecución de transacciones en un entorno de datos distribuidos, indicando para cada una de ellas el protocolo de commit atómico (ACP) a utilizar para la recuperación ante distintos casos de falla que se presentan a continuación, manteniendo en todo momento la consistencia de la base de datos. Dicho ambiente se implementó en Java.

Las distintas *situaciones de fallo* que pueden ocurrir durante la ejecución de una transacción distribuida son:

- *Fallo de una localidad participante.* Para recuperarse, debe examinar su bitácora y a partir de allí, decidir el destino de la transacción.
- *Fallo del coordinador.* En este caso, las localidades participantes son quienes deben decidir el destino de la transacción. Si no poseen suficiente información, tendrán que esperar a que se recupere el coordinador.

La implementación del ambiente se realizó en Java. Se utilizó el JDK 1.3, conjuntamente con el Forte For Java 1.0 para construir la interface con el usuario (GUI). De esta forma se logra contar con interoperabilidad relacionada al hardware, permitiendo que cualquier computadora que posea JVM (Java Virtual Machine) pueda ejecutar el ambiente sin ningún inconveniente.

La utilización de JAVA presentó ventajas y mejoras. Estas mejoras están principalmente relacionadas con el ambiente de simulación, el cual presenta una interface amigable. Además, Java permite manipular conexiones vía JDBC (Java

DataBase Connectivity) con la base de datos, el cual representa una ventaja para el desarrollador del ambiente en comparación con otras herramientas que actualmente se hallan en el mercado.

Consideremos un sistema distribuido compuesto por un conjunto finito de sitios completamente conectados a través de un conjunto de canales de comunicación. Cada sitio posee memoria local y ejecuta uno o varios procesos. Por simplicidad, se asume solo un proceso por sitio. Los procesos se comunican entre sí intercambiando mensajes de forma asincrónica. En un momento determinado, un proceso puede estar en alguno de los siguientes dos estados: “Operacional” o “En Fallo”.

En el estado “Operacional”, un proceso seguirá con las acciones especificadas por la transacción que está ejecutando. Un proceso operacional puede fallar, pasando de esta manera al estado “En Fallo”. Consideraremos que un proceso que se encuentra en este estado, en algún momento retornará al estado operacional.

Un proceso que falla detiene todas sus actividades, incluyendo el envío de mensajes a otros procesos, hasta el momento que se recupera y vuelve al estado operacional . Cada uno de ellos posee acceso al almacenamiento estable (bitácora), en donde se mantiene la información necesaria para el protocolo de recuperación. Esto significa que, durante la recuperación, el proceso restablece su estado normal usando la información de la bitácora. Además, ante una instrucción de modificación de datos se opera con la técnica de modificación inmediata de los datos, debiendo almacenar el valor viejo y el valor nuevo.

Otra de las suposiciones que se tienen en cuenta es que si un mensaje es enviado desde el proceso P_i a P_k , es eventualmente recibido por P_k , siempre y cuando P_i y P_k no se encuentran "en fallo".

Se dispone de cuatro clases diferentes de tareas:

- Transaction Manager
- Transaction Server
- Coordinator
- Agent

Los procesos que simulan al coordinador y a los agentes participantes de la transacción serán “Threads” que son creados por los procesos “Transaction Server” que residen en cada localidad.

En un ambiente de base de datos distribuida, una transacción puede acceder a datos que están almacenados en más de un sitio de la red. Cada transacción es dividida en una serie de sub-transacciones, una por cada sitio en donde existan datos que la transacción original necesita procesar. Estas sub-transacciones son implementadas por los procesos “Agent” que representan cada localidad de la BDD.

Inicialmente, se ejecuta el proceso “Transaction Manager”, el cual provee la interface para que el usuario pueda realizar cualquier tipo de operación sobre la base de datos. Esto significa la realización de un Query, el cual puede ser tanto una consulta como una operación de inserción, borrado o actualización. Para realizar estas operaciones, el usuario no debe ingresar directamente código SQL, sino que, a través de una interface, se define la información que será utilizada por el ambiente para generar la operación que posteriormente será ejecutada en la base de datos. Esta definición incluye principalmente las tablas sobre las que se va a operar, el tipo de operación a realizar, sitio en donde se desea iniciar la ejecución de la transacción, etc. Esta última elección permite al usuario de la simulación, que una misma operación sobre la BDD se inicie en diferentes lugares y para poder, luego, obtener y comparar los resultados, teniendo en

cuenta la distribución inicial de los datos en cada localidad. El sitio que origina una transacción actúa como el coordinador de la misma, y como tal, es el encargado de tomar todas las decisiones respecto de su ejecución.

Un parámetro a definir, para llevar a cabo la simulación, consiste en identificar el protocolo de commit que se llevará a cabo durante la ejecución de las transacciones. Esto permitirá que para una misma traza de ejecución, se pueda observar como responde el prototipo ante distintos protocolos para poder realizar una comparación de los resultados obtenidos.

Cada localidad habilitada para la intervención en la ejecución de una transacción dentro del ambiente de simulación posee un archivo de configuración (aún estático) que permite conocer la ubicación y el uso de cada tabla del entorno distribuido. A partir de esta información, el coordinador determinará cuáles serán las localidades que participarán en la ejecución de las sub-transacciones que se generen a partir de la transacción a simular.

Como los casos de estudios elegidos para esta etapa del entorno de simulación contemplan un estudio del comportamiento ante fallos, se deberá indicar en la trazqa de ejecución, cual será la localidad (o el coordinador) que sufrirá el fallo, produciéndose así el entorno bajo el cual se simula el protocolo de recuperación de la base de datos seleccionado. En el caso de estudio se contempla la simulación de una caída en la línea de comunicaciones produciendo solamente el fallo en la localidad conectada a través de esa línea.

Entorno de trabajo

Las primeras experiencias realizadas con el entorno de simulación definido utilizaron una red local, es nuestra intención llevar esta simulación hacia una red distribuida geográficamente, debiendo contar solamente con estaciones de trabajo para lograr este objetivo sin necesidad de hacer cambios en el entorno definido. Esto se debe a que crear los sockets basta con indicar la dirección física de cada localidad y posteriormente ejecutar en cada una de ellas procesos Server.

A fin de lograr el entorno de trabajo, la ubicación física de las localidades es totalmente transparente, por este motivo se puede solucionar el problema de la distribución geográfica indicando como parámetro de simulación el “costo” asociado a una transmisión entre dos localidades, independizándose de esquema de red utilizado.

Modelo de Simulación

Las transacciones ejecutadas pueden ser definidas como locales o globales (normalmente respetando una proporción 80-20). En este último caso se define para dicha transacción el número de sub-transacciones que la componen, y que involucran, a priori, diferentes localidades donde se ejecutarán. Es necesario que cada localidad simulada esté ejecutando un proceso denominado “Transaction Server”. Además, desde el sitio donde se comienza la ejecución de la transacción se debe ejecutar un proceso “Transaction Manager”.

El proceso “Transaction Manager”, a partir de las especificaciones de las transacciones (las cuales pueden estar en un archivo de trazas o definirse on line), determina qué proceso “Transaction Server” residente en alguna localidad del entorno de simulación, será el Coordinador de la transacción. A continuación, este thread

determina, consultando la tabla de ubicaciones, quiénes serán las localidades intervinientes y se lo comunica a los “Transaction Server” correspondientes.

Cuando un “Transaction Server” recibe un mensaje de un coordinador indicándole la participación como localidad en la ejecución de la transacción, crea un thread “Agent”.

A partir del establecimiento de los canales de comunicación entre la localidad generadora de la transacción (la cual actúa como coordinadora) y cada uno de las localidades intervinientes se comienza la ejecución de la transacción distribuida. Cada “Agent” ejecuta su parte de la transacción y al finalizar indica al coordinador dicho estado. Este coordinador implementa, como se dijo anteriormente, el protocolo de commit atómico (ACP) elegido al principio de la simulación, para mantener íntegra la BDD.

Condiciones a Comparar

Desde el punto de vista de la performance, los protocolos de commit pueden ser comparados teniendo en cuenta lo siguiente:

- **Efecto sobre el procesamiento normal:** Se refiere a como el protocolo afecta a la performance de procesamiento de la transacción distribuida normal (sin fallas). Esto es, cuánto nos cuesta proveer atomicidad usando este protocolo?
- **Flexibilidad en fallos:** Un protocolo de commit es *no bloqueante* si, en el caso de que un sitio falle, permite que los otros sitios sigan ejecutando sin tener que esperar a que el que falló se recupere. Para permitir esta funcionalidad, se incurre usualmente al pasaje de mensajes adicionales, a los que se utilizan en los protocolos *bloqueantes*. En general, el *protocolo de commit dos fases* es *bloqueante* mientras que el *de tres fases* es *no bloqueante*.
- **Velocidad de Recuperación:** Se refiere al tiempo requerido por la base de datos para recuperarse ante un fallo de un sitio. Esto es, cuánto tiempo pasó, antes que el procesamiento de la transacción pueda comenzar nuevamente, en el sitio que se acaba de recuperar?

Los primeros dos son los más importantes, ya que afectan directamente al procesamiento de la transacción. En comparación, el último punto, aparece menos crítico por diversas razones, entre otras, la duración de una falla posee usualmente un orden de magnitud mayor al tiempo de recuperación del sitio. Desde este punto de vista, es más importante enfocarse en los mecanismos requeridos durante la operación normal de recuperación, en vez del tiempo de recuperación mismo.

Comparación de protocolos

El mejor protocolo de commit atómico (ACP) es el de dos fases (2PC), el cual fue adoptado por la mayoría de los standart transaccionales, como por ejemplo DTP de X/OPEN y OTS de OMG, como así también implementado en varios sistemas de bases de datos comerciales. A pesar de ésto, 2PC es considerado un poco ineficiente ya que introduce un “substancial delay” (bloqueo) en el procesamiento de una transacción. Este delay está supeditado al costo de tiempo asociado con la coordinación entre mensajes y las escrituras forzadas en la bitácora.

Supongamos que *n* es el número total de participantes (incluyendo al coordinador), 2PC requiere tres etapas de comunicación (el requerimiento para el "voto", el "voto" y la "decisión") y 2*n* + 1 escrituras forzadas en almacenamiento estable, para una transacción con ausencia de fallos. Esto introduce una considerable latencia en el sistema, la cual incrementa el tiempo de ejecución de la transacción.

La probabilidad de que en la práctica ocurra un bloqueo es lo suficientemente baja para que no esté justificado el coste extra que supone el commit de tres fases. Además, el commit de tres fases es muy vulnerable a la hora de enlazar fallos. Esta desventaja puede ser salvada con protocolos de nivel de red, pero de esta forma se aumenta el tiempo extra.

Ambos protocolos pueden ser perfeccionados para reducir el número de mensajes enviados y para reducir el número de veces que los registros son grabados en almacenamiento estable.

Estas limitaciones han incentivado a varios investigadores a trabajar en versiones optimizadas o alternativas de 2PC. Una de ellas es el "Protocolo Optimista" (PrC) y otra el "Protocolo Pesimista" (PrA). Estos protocolos reducen el costo de 2PC en términos de escrituras en bitácora y complejidad de mensajes.

La latencia de un ACP está determinada por la cantidad de escrituras de bitácora forzadas y pasos de comunicación llevados a cabo durante la ejecución del protocolo, y hasta que se llegue a una decisión en cada participante. La Figura 1 compara los diferentes protocolos en términos de latencia y complejidad del mensaje necesarios para encomendar/cometer (commit) una transacción. Al comparar con el 2PC, el PrA no reduce el costo de encomendar/cometer (commiting) las transacciones. El PrC requiere menos mensajes y escrituras de bitácora forzadas que el 2PC pero no reduce la cantidad de pasos de comunicación requeridos para encomendar/cometer (commit) una transacción.

Protocolo de Commit Atómico (ACP)	Número de Mensajes	Latencia	Latencia
		Escrituras en Bitácora	Número de etapas de comunicación
2PC	4(*n* - *1*)	1 + 2*n*	3
PrA	4(*n* - *1*)	1 + 2*n*	3
PrC	3(*n* - *1*)	2 + *n*	3

Figura 1

Conclusiones

El objetivo perseguido es definir, modelar e implementar un ambiente de simulación para el mantenimiento y recuperación de datos, en un sistema de BDD, donde se pueda estudiar, monitorear y posteriormente comparar resultados a partir de los distintos protocolos de commit elegidos para una misma transacción.

Las trazas de ejecución probadas hasta el momento nos demuestran que las variantes del 2PC (PrC y PrA) reducen el overhead del protocolo y proveen una mejora con respecto a 2PC solo en situaciones muy puntuales. PrC presenta una mejor performance cuando el grado de distribución de los datos es alto, pero en las aplicaciones actuales, el grado de distribución es usualmente bajo. Por otro lado, PrA ofrece una ventaja con respecto al 2PC cuando la probabilidad de transacciones que aborten es baja.

En resúmen, en un ambiente de BDD es aconsejable utilizar el 2PC, PrC o PrA, ya que ofrecen un mejor beneficio que el 3PC. Pero en el caso en que se necesite que no se produzcan los “bloqueos”, como en el 2PC, el protocolo de tres fases aparece como el más atractivo.

Bibliografía

[BELL 92] *Distributed Database Systems*, Bell, David; Grimson, Jane. Addison Wesley. 1992

[BERN 84] *An algorithm for concurrency Control and Recovery in Replicated Distributed Databases* Bernstein, Goodman. ACM Trans. Database Systems, vol 9 no. 4, pp. 596-615, Dec 1984.

[BERS 92] *Client Server Architecture.* Berson, Alex. Mc Graw Hill Series. 1992

[BERT 99] *Ambiente de experimentación para Bases de Datos Distribuidas*. Bertone, Rodolfo; De Giusti, Armando; Ardenghi, Jorge. Anales WICC. Mayo 1999. San Juan Argentina.

[BHAS 92] *The architecture of a heterogeneous distributed database management system: the distributed access view integrated database (DAVID).* Bharat Bhasker; Csaba J. Egyhazy; Konstantinos P. Triantis. CSC '92. Proceedings of the 1992 ACM Computer Science 20th annual conference on Communications, pages 173-179

[BURL 94] *Managin Distributed Databases. Building Bridges between Database Island.* Burleson, Donal. 1994

[DATE 94] *Introducción a los sistemas de Bases deDatos*. Date, C.J. Addison Wesley 1994.

[DIPA 99] *Un ambiente experimental para evaluación de Bases de Datos Distribuidas.* Di Paolo, Mónica; Bertone, Rodolfo; De Giusti, Armando. Paper presentado (aún no evaluado) en la ICIE 99. Fac. Ingeniería. UBA. Buenos Aires. Argentina

[LARS 95] *Database Directions. From relational to distributed, multimedia, and OO database Systems.* Larson, James. Prentice Hall. 1995

[MIAT 98] *Experiencias en el análisis de fallas en BDD.* Miaton, Ivana; Ruscuni, Sebastián; Bertone, Rodolfo; De Giusti, Armando. Anales CACIC 98. Neuquén Argentina.

[ÖZSU 91] *Principles of Distributed Database Systems.* Özsu, M. Tamer; Valduriez, Patric. Prentice Hall 1991.

[SCHU 94] *The Database Factory. Active databse for enterprise computing.* Schur, Stephen. 1994.

[SHET 90] *Federated database systems for managing distributed, heterogeneous, and autonomous databases.* Amit P. Sheth; James A. Larson. ACM Computing Surveys.Vol. 22, No. 3 (Sept. 1990), Pages 183-236

[SILB 98] *Fundamentos de las Bases de Datos.* Silbershatz; Folk. Mc Graw Hill. 1998.

[THOM 90] *Heterogeneous distributed database systems for production use.* Thomas, Charles; Glenn R. Thompson; Chin-Wan Chung; Edward Barkmeyer; Fred Carter; Marjorie Templeton; Stephen Fox; Berl Hartman. ACM Computing Surveys. Vol. 22, No. 3 (Sept. 1990), Pages 237-266

[UMAR 93] *Distributed Computing and Client Server Systems.* Umar, Amjad. Prentice Hall. 1993.

[WEB1] *www.seas.gwu.edu/faculty/shmuel/cs267/textbook/tpcp.html*

[WEB2] *www.sei.cmu.edu/str/descriptions/dtpc_body.html*

[WEB3] *www.datanetbbs.com.br/dclobato/textos/bddcs/parte2.html*

[WEB4] *www.itlibrary.com/library/1575211971/ch45.htm*

[WEB5] *www.itlibrary.com/library/1575211971/ch26.htm*

[WEB6] *java.sun.com/docs/books/tutorial/networking/sockets/definition.html*

[ZANC96] *Análisis de Replicación en Bases de Datos Distribuidas*, Marcelo Zanconi, Tesis de Magister en Ciencias de la Computación, Univ. Nac. Del Sur, Bahía Blanca, 1996